A Constraint for Bin Packing

Paul Shaw

ILOG S.A., Les Taissounieres HB2
2681 Route des Dolines, 06560 Valbonne, France
pshaw@ilog.fr

Abstract. We introduce a constraint for one-dimensional bin packing. This constraint uses propagation rules incorporating knapsack-based reasoning, as well as a lower bound on the number of bins needed. We show that this constraint can significantly reduce search on bin packing problems. We also demonstrate that when coupled with a standard bin packing search strategy, our constraint can be a competitive alternative to established operations research bin packing algorithms.

1 Introduction

The one-dimensional bin packing problem is described as follows: Given n indivisible items, each of a known non-negative size s_i, and m bins, each of capacity C, can we pack all n items into the m bins such that the sum of the sizes of the items in any bin is not greater than C? This problem is an important NP-complete problem having various applications (a good review paper [2] cites stock cutting and television commercial break scheduling as well as the obvious application of physical packing).

In recent years, good exact algorithms for the bin packing problem have been put forward (for example, see [10, 11, 15]), so why examine a constraint-based approach? The main reason is that most real bin packing problems are not pure ones, but form a component of a larger system. For example, almost all resource allocation problems have a bin packing component; this set includes timetabling, rostering, scheduling, facility location, line balancing, and so on. When problems become less pure, standard algorithms often become inapplicable, making constraint programming more attractive. Additionally, the use of a dedicated constraint (in place of a collection of constraints defining the same solution set) normally allows for a significant increase in constraint propagation, the all-different constraint [14] being the classic example.

In this paper, we introduce a dedicated constraint $\mathcal{P}$ for the one-dimensional bin packing problem. Section 2 introduces some notation as well as a typical bin packing model. Section 3 describes new pruning and propagation rules based on reasoning over single bins. Section 4 then introduces a lower bounding method on the number of bins used. Section 5 describes experiments on the new constraint. Finally, section 6 describes the most relevant work and section 7 concludes.

M. Wallace (Ed.): CP 2004, LNCS 3258, pp. 648–662, 2004.
© Springer-Verlag Berlin Heidelberg 2004

2 Preliminaries

2.1 General Notation

All constrained variables we consider in this paper are non-negative integer. Associated with each variable x is an initial domain $D_0(x) \subset \{0 \dots \infty\}$. The goal is to assign each variable an element from its domain without violating any constraints: such an assignment is referred to as a *solution*. We assume that this assignment procedure proceeds constructively, building the solution one piece at a time. At each point in this construction, we have a *partial assignment* which we define to be the set of *current domains* of all variables. The current domain $D(x)$ of variable x is always a (non-strict) subset of its initial domain $D_0(x)$. We also denote the minimum and maximum of the domain of x in the current partial assignment as $\underline{x}$ and $\overline{x}$ respectively.

When performing a domain reduction on a variable x, assume that $D'(x)$ is the domain of x after the reduction. We use $x \leftarrow a$ to denote $D'(x) = D(x) \cap a$, $\underline{x} \leftarrow a$ to denote $D'(x) = D(x) \cap \{a \dots \infty\}$, $\overline{x} \leftarrow a$ to denote $D'(x) = D(x) \cap \{0 \dots a\}$, and $x \not\leftarrow a$ to denote $D'(x) = D(x) \backslash a$.

If at any time, for any variable x, $D(x) = \emptyset$, then the constraint system prunes the search. What happens then depends on the search strategy employed, but normally the search will backtrack to a previous choice point (if one exists), changing an earlier decision.

2.2 Bin Packing Problem

Any instance of the packing constraint $\mathcal{P}$ takes three parameters which are a vector of m constrained variables $l = \langle l_1 \dots l_m \rangle$ representing the *load* of each bin, a vector of n constrained variables $b = \langle b_1 \dots b_n \rangle$ indicating, for each item, the index of the bin into which it will be placed, and a vector of n non-negative integers $s = \langle s_1 \dots s_n \rangle$ representing the size of each item to be packed. Without loss of generality, we assume that the sizes are sorted according to $s_i \geq s_{i+1}$. The total size to be packed is denoted by $S = \sum_{i=1}^{n} s_i$. The set of item indices is represented by $I = \{1 \dots n\}$ and the set of bin indices by $B = \{1 \dots m\}$.

Note that, because variables in b and l can have arbitrary domains when given to the constraint, problems more general that pure bin packing problems can be specified. These include variable-sized bin packing [4], problems where bins have a minimum load requirement, and those where not all items can be packed in all bins.

The semantics of the bin packing problem dictate that $\mathcal{P}$ must ensure for each bin $j \in B$ that $l_j = \sum_{\{i \mid i \in I \wedge b_i = j\}} s_i$ and for each item $i \in I$ that $b_i \in B$.

2.3 Bin Packing Notation

We introduce some notation to specify the states of bins in partial assignments. We define as the *possible set* P_j of a bin j as the set of items that are packed or may potentially be packed in it: $P_j = \{i \mid i \in I \wedge j \in D(b_i)\}$. We define the

required set R_j of bin j as the set of items packed in the bin: $R_j = \{i \mid i \in P_j \wedge |D(b_i)| = 1\}$. We refer to $C_j = P_j - R_j$ as the *candidate set* of bin j. We refer to the total size of items packed in bin j as $p_j = \sum_{i \in R_j} s_i$, and to the set of unpacked items as $U = \{i \mid i \in I \wedge |D(b_i)| > 1\}$.

2.4 Typical Bin Packing Model

We present what we consider to be a typical constraint programming model of the bin packing problem, which will be referred to as a comparison base in the rest of the paper. We introduce intermediate 0–1 variables $x_{i,j}$ that determine if item i has been placed in bin j. These variables are maintained by nm constraints as below. We have, for each item i:

$$\forall j \in B \; x_{i,j} = 1 \Leftrightarrow b_i = j$$

Alternatively, we could remove the b variables from the model and add n constraints of the form $\sum_{j \in B} x_{i,j} = 1$. The x variables would then become the decision variables.

The loads on the bins are maintained by m scalar products. For bin j we have:

$$l_j = \sum_{i \in I} w_i x_{i,j}$$

All items must be packed, and for each item i:

$$b_i \in B$$

Finally, it is often useful to add the redundant constraint specifying that the sum of the bin loads is equal to the sum of the item sizes.

$$\sum_{j \in B} l_j = S$$

2.5 Propagations Performed Both by the Typical Model and by $\mathcal{P}$

When the typical model in the previous section is implemented in ILOG Solver [7] it carries out certain "basic" constraint propagations. These propagations are also carried out by $\mathcal{P}$, and each one of them is detailed in this section.

Pack All. All items must be packed. This is enforced for each item $i \in I$ via:

$$\underline{b_i} \leftarrow 1 \qquad\qquad \overline{b_i} \leftarrow m$$

Load Maintenance. The minimum and maximum load of each bin is maintained according to the domains of the bin assignment variables b. For brevity, we denote $\textsc{Sum}(X) = \sum_{i \in X} s_i$, where $X \subseteq I$ is any set of items under consideration. For each bin $j \in B$:

$$\underline{l_j} \leftarrow \textsc{Sum}(R_j) \qquad\qquad \overline{l_j} \leftarrow \textsc{Sum}(P_j)$$

Load and Size Coherence. We perform the following propagations which are derived from the fact that the sum of the items sizes must be equal to the sum of bin loads. This means that for any bin $j \in B$, its load is equal to the total size to be packed, minus the loads of all other bins. This translates into the following propagation rules:

$$\underline{l_j} \leftarrow S - \sum_{k \in B \setminus j} \overline{l_k} \qquad\qquad \overline{l_j} \leftarrow S - \sum_{k \in B \setminus j} \underline{l_k}$$

This propagation rule is very important as it is only rule which communicates information between different bins (aside from the implicit rule disallowing item i from all bins other than j when item i is packed in bin j.) Especially important is that it can increase the *lower bound* of the bin load. This property is analogous to the notion of spare capacity used in [5]. The spare capacity sc in a bin packing problem is $sc = mC - S$ (the capacity available less the total size to be packed). In any solution, each bin must be filled to at least a level of $C - sc$, otherwise more space would be wasted than spare capacity available. The above propagation rules dynamically maintain this information, propagating bounds on load and implicitly reducing spare capacity when a bin is packed to less than its full capacity.

The commitment rule below makes use of the lower bound on bin load, as do the additional pruning and propagation rules introduced in section 3.

Single Item Elimination and Commitment. An item is eliminated as a candidate for packing in a bin if it cannot be added to the bin without the maximum load being exceeded. For bin j:

$$\text{if } \exists i \; i \in C_j \wedge p_j + s_i > \overline{l_j} \text{ then } b_i \not\leftarrow j$$

An item is committed to a bin if packing all candidates into the bin except that item would not increase the packed quantity to the required minimum load of the bin. For bin j:

$$\text{if } \exists i \; i \in C_j \wedge \textsc{Sum}(P_j) - s_i < \underline{l_j} \text{ then } b_i \leftarrow j$$

3 Additional Propagation Rules

The idea of constraint propagation is to eliminate domain values when it can be ascertained that these values can never appear in a solution to the constraint. (A solution to a constraint is an assignment of all variables involved in the constraint which does not violate the constraint.) Constraint propagation algorithms often try to achieve generalized arc consistency (GAC) which means that *all* values which cannot be involved in a solution to the constraint are removed. Unfortunately, for the packing constraint $\mathcal{P}$, achieving GAC is NP-complete. So, instead of trying to achieve GAC, we set ourselves the more humble goal of increasing propagation strength over that of the typical model.

The additional constraint propagation rules that we introduce are all based upon treating the simpler problem of packing a single bin. That is, given a bin j, can we find a subset of C_j that when packed in the bin would bring the load the range $[\underline{l_j}, \overline{l_j}]$? This problem is a type of knapsack or subset sum problem [9]. Proving that there is no solution to this problem for any bin j would mean that search could be pruned. However, as shown by Trick [17], we can go further and even achieve GAC for this reduced single-bin problem.

For bin j we define $M_j = \{m \mid m \subseteq C_j \wedge \underline{l_j} \leq p_j + \textsc{Sum}(m) \leq \overline{l_j}\}$. M_j is the set of sets of additional items which can be packed while respecting the constraints on load. If M_j is empty, there is no legal packing of bin j, and we can prune search. For $M_j \neq \emptyset$, we make deductions on the candidate items:

$$\text{if } \forall m \in M_j \ i \in m \text{ then } b_i \leftarrow j \qquad \text{if } \forall m \in M_j \ i \notin m \text{ then } b_i \not\leftarrow j$$

That is, if an item appears in every set of items that can be placed in the bin, we can commit it to the bin. Conversely, if the item never appears in such a set, we can eliminate it as a candidate item. Trick did not treat the case where the load was a variable, but it is nevertheless possible to deduce illegal bin loads:

$$\text{if } \exists v \in D(l_j) \ \forall m \in M_j \ v \notin m \text{ then } l_j \not\leftarrow v$$

So, if no legal packing can attain load v, v cannot be a legal load for the bin.

In [17], a pseudo-polynomial dynamic programming algorithm is used to achieve consistency. The time complexity of the algorithm is $O(|C_j|\overline{l_j}^2)$, meaning that it can become inefficient when items are large or many items can be packed in a bin. Sellmann [16] proposes an approach where efficiency can be traded for propagation strength by dividing down values; in our case, item sizes and bin capacities. By selecting an appropriate divisor, the resulting algorithm can produce a trade-off between propagation strength and time invested. Here, we take a different approach, using efficient algorithms which do not depend on item size or bin capacity, but which like [16], do not in general achieve GAC on the subset sum subproblem.

3.1 Detecting Non-packable Bins

Here, we describe a method which can detect non-packable bins in time $O(|C_j|)$. The advantage of the method is its efficiency; the drawback is that it is not complete, and some non-packable bins may not be identified as such.

We try to find a proof that $\forall m \subseteq C_j \ p_j + \textsc{Sum}(m) < \underline{l_j} \vee p_j + \textsc{Sum}(m) > \overline{l_j}$. That is, there is no subset of candidate items whose sizes – together with the already packed items – sum to a legal load. We search for this proof in a particular way based on identifying what we refer to as *neighboring subsets*[1] of C_j. Assume two sets $C_j^1, C_j^2 \subseteq C_j$. These sets are neighboring if there is no other subset of C_j whose items sum to a value strictly between $\textsc{Sum}(C_j^1)$ and $\textsc{Sum}(C_j^2)$. (This

[1] The author has found no pre-existing definition of the proposed notion in the literature.

implies that C_j^1 and C_j^2 are neighboring if $|\textsc{Sum}(C_j^2) - \textsc{Sum}(C_j^1)| \leq 1$.) We describe a neighboring predicate $N(j, C_j^1, C_j^2)$ as follows:

$$N(C_j^1, C_j^2) \Leftrightarrow N(C_j^2, C_j^1)$$

$$\textsc{Sum}(C_j^1) \leq \textsc{Sum}(C_j^2) \Rightarrow (N(C_j^1, C_j^2) \vee \textsc{Between}(j, C_j^1, C_j^2))$$

$$\textsc{Between}(j, C_j^1, C_j^2) \Leftrightarrow \exists m \subseteq C_j \ \textsc{Sum}(C_j^1) < \textsc{Sum}(m) < \textsc{Sum}(C_j^2)$$

Generation of Neighboring Subsets. Neighboring subsets are generated which conform to a particular structure. We use the terms *low-set* and *high-set* for two subsets which we consider as candidates for being neighbors. The understanding is that when the low-set has a sum not greater than the high-set, then the two subsets are neighbors. This will become clearer in what follows.

Figure 1 shows a set of candidate items X sorted by non-increasing size, and three subsets marked A, B and C. Subset A comprises the k largest candidate items. Subset C comprises the k' smallest candidate items. Subset B comprises the $k+1$ smallest items outside subset C. Subsets A and B may contain the same item, but subset C is disjoint from the other two. We choose the low-set L_k to be the union of subsets A and C and the high-set H_k to be the subset B.

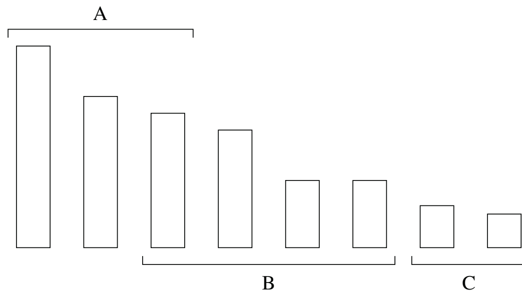

Fig. 1. Structure of neighboring subsets

We now show that for this particular structure, if $\textsc{Sum}(L_k) \leq \textsc{Sum}(H_k)$, then L_k and H_k are neighboring. When $\textsc{Sum}(L_k) = \textsc{Sum}(H_k)$, the proof is trivial. We therefore concentrate on the case where $\textsc{Sum}(L_k) < \textsc{Sum}(H_k)$. L_k is formed from the k largest items and the k' smallest items. We can see that for any $m \subseteq X$ for which $\textsc{Sum}(m) > \textsc{Sum}(L_k)$, $|m| \geq k+1$. This must be the case as L_k already contains the k largest items. Moreover m cannot contain any of the k' smallest items as then $\textsc{Sum}(L_k) \geq \textsc{Sum}(H_k)$ in contradiction to our initial assumption. To see this, imagine $i \in m$ and that i is one of the smallest k' items in X. In this case m is composed of k items not in the smallest k' and item i, while L_k is composed of the largest k items, plus item i (and possibly some other items). Since the largest k items must have a total size not less than k items chosen more freely, it follows that if m contains one of the k' smallest items then

```
boolean function NoSum(X, α, β)
  if α ≤ 0 ∨ β ≥ Sum(X) then
    return false
  Σ_A, Σ_B, Σ_C := 0   {See figure 1 for meaning of A, B, C}
  k, k' := 0      {k largest items, k' smallest items}
  while Σ_C + s_{|X|-k'} < α do
    Σ_C := Σ_C + s_{|X|-k'}
    k' := k' + 1
  end while
  Σ_B := s_{|X|-k'}
  while Σ_A < α ∧ Σ_B ≤ β do
    k := k + 1
    Σ_A := Σ_A + s_k
    if Σ_A < α then
      k' := k' - 1
      Σ_B := Σ_B + s_{|X|-k'}
      Σ_C := Σ_C - s_{|X|-k'}
      while Σ_A + Σ_C ≥ α do
        k' := k' - 1
        Σ_C := Σ_C - s_{|X|-k'}
        Σ_B := Σ_B + s_{|X|-k'} - s_{|X|-k'-k-1}
      end while
    end if
  end while
  return Σ_A < α
end function
```

Fig. 2. Procedure for detecting non-existence of sums in $[\alpha, \beta]$

$\text{Sum}(m) \leq \text{Sum}(L_k)$. Having now discounted the smallest k' items from m, the smallest $\text{Sum}(m)$ is then obtained when m is made up of the $k+1$ smallest items outside of the smallest k', *i.e.* when $m = H_k$.

As an example of this reasoning, we consider a bin j with $p_j = 0$, $\underline{l_j} = 34$, $\overline{l_j} = 35$ and candidates items of sizes 10, 10, 10, 9, 9, 9, 9, 2, 1. The bin can be shown non-packable by considering the neighboring subsets produced when $k = 3$ and $k' = 2$. In this case, the low-set sums to $10+10+10+2+1 = 33$ and the high-set to $9+9+9+9 = 36$.

Implementation. Here we describe a procedure NoSum which determines if a subset of a set of items X cannot sum to a value in the range $[\alpha, \beta]$. Being an incomplete method, NoSum can generate false negatives but this is not problematic as only the positive result is actively used in pruning. We remind the reader that item sizes obey the relation $s_i \geq s_{i+1}$. NoSum is detailed in figure 2.

The method is reasonably simple despite its appearance. The variables Σ_A, Σ_B and Σ_C hold the total sizes of items in the subsets A, B, and C shown in figure 1. The variables k and k' keep their meaning from the previous section.

Essentially, k is increased from 0 until the k largest items are not less than α, the lower bound on the required sum. For each k, k' is chosen such that $\Sigma_A + \Sigma_C$ is maximized while being less than α. This is done initially by adding up the k' smallest item sizes. Thereafter, each time k is increased, k' is reduced again until $\Sigma_A + \Sigma_C < \alpha$. During this reduction phase, Σ_B is maintained by 'sliding' the window of the B subset to the right so that it remains adjacent to subset C (again, see figure 1). If we find that $\Sigma_B > \beta$, then we have found a proof that no subset of the items can sum to a value in $[\alpha, \beta]$.

The time complexity of NoSum is linear in $|X|$, but is usually much better as it is bounded by the value of k' after the initial loop. That is, the maximum number of candidate items that can be added together while maintaining their sum strictly less than α. We refer to this quantity as N_j^α. We assume that Sum(X) can be computed in constant time by incrementally maintaining the total item size of any candidate set when it is reduced.

3.2 Pruning Rule

Given the function NoSum, it is trivial to derive a pruning rule. The only manipulation needed is to remove the effect of items already packed in the bin from the load target. For any bin j:

$$\text{if } \textsc{NoSum}(C_j, \underline{l_j} - p_j, \overline{l_j} - p_j) \text{ then } \textbf{prune}$$

3.3 Tightening Bounds on Bin Load

The reasoning used for detecting non-packable bins can also be used to tighten bounds on the bin load. For this, NoSum needs slight modification to deliver on return the values of $\alpha' = \Sigma_A + \Sigma_C$ and $\beta' = \Sigma_B$. These are the total sizes of the two neighboring subsets when NoSum answers in the affirmative. We can then specify the tightening rules as:

$$\text{if } \textsc{NoSum}(C_j, \underline{l_j} - p_j, \underline{l_j} - p_j) \text{ then } \underline{l_j} \leftarrow p_j + \beta'$$

$$\text{if } \textsc{NoSum}(C_j, \overline{l_j} - p_j, \overline{l_j} - p_j) \text{ then } \overline{l_j} \leftarrow p_j + \alpha'$$

3.4 Elimination and Commitment of Items

As proposed in [5], we construct from the pruning rule propagation rules to eliminate and commit items. We make a proposition, then ask the pruning rule if this proposition leads to a contradiction; if so, we can assert the negation of the proposition. We commit item i to bin j and then ask if bin j is packable. If it is not, then i can be eliminated from the candidates of bin j. A similar argument holds for committing items; we assume that i is eliminated, then ask if the bin is packable. If not, we can commit the item to the bin. Given a candidate item $i \in C_j$:

$$\text{if } \textsc{NoSum}(C_j \backslash i, \underline{l_j} - p_j - s_i, \overline{l_j} - p_j - s_i) \text{ then } b_i \not\leftarrow j$$

$$\text{if } \textsc{NoSum}(C_j \backslash i, \underline{l_j} - p_j, \overline{l_j} - p_j) \text{ then } b_i \leftarrow j$$

In both of these rules, item i is eliminated as a candidate, but in the first, the bounds passed to NoSum are reduced to reflect the increased size of the packed items due to the inclusion of i in R_j. In the second rule, the proposition is to eliminate an item and so the packed size does not change.

Examining a bin j for all item eliminations and commitments takes time $O(|C_j|N_j^{l_j})$, which can be up to $O(|C_j|^2)$ but is typically much less as normally $N_j^{l_j} \ll |C_j|$. When the pruning rule is restricted to consider only neighboring subsets for which either $k = 0$ or $k' = 0$, we know of algorithms which run in time $O(|C_j|)$. These algorithms combine the logic of the pruning and propagation in one procedure and are thus more complex than the use of the pruning rule as a subordinate procedure. In this paper, these linear algorithms are not used do to their increased complexity, reduced propagation strength (as the pruning rule used is less general), and marginal reduction in calculation time per search node.

4 Using a Lower Bound

The usual simple lower bound for the fixed capacity bin packing problem is arrived at by relaxing the constraint that each item is indivisible. Given a fixed capacity C, the bound, which we shall term L_1 (following Martello and Toth [13]) is:

$$L_1(C, s) = \left\lceil \frac{1}{C} \sum_{i \in I} s_i \right\rceil$$

Martello and Toth showed L_1 to have an asymptotic performance ratio of $\frac{1}{2}$, which means that the the bound L_1 can be arbitrarily close to one half of the number of bins used in an optimal solution.

However there are better bounds which also are efficient to compute. In [13], Martello and Toth introduced bound L_2 which dominates L_1 and has an asymptotic performance ratio of $\frac{2}{3}$. In their experiments, Martello and Toth show this bound to be typically much tighter than L_1. Korf [10] also found this. In his tests on problems with random sizes from 0 up to the bin capacity, 90 item problems require an average of 47.68 bins. For these problems, the L_1 bound averaged 45.497, whereas the L_2 bound averaged 47.428. The L_2 bound can be calculated in linear time if the item sizes are sorted.

The bound works by splitting the items into four subsets using a parameter K. N_1 contains all items that are strictly larger than $C - K$. N_2 contains all items not in N_1, but which are strictly larger than half the bin capacity. N_3 contains all items not in N_1 or N_2 of size at least K. The items of size strictly less than K are not considered. Now, no items from N_2 or N_3 can be placed with items from N_1, so $|N_1|$ forms a term of the lower bound. Moreover, each item in $|N_2|$ needs a bin of its own and so $|N_2|$ forms another term. Finally, the free space in the bins of N_2 is subtracted from the total size of items in N_3 (as items in N_3 can be placed with those in N_2). If the result is positive, this is an overspill which will consume more bins. The result of this extra bin calculation is added to $|N_1| + |N_2|$ to arrive at the bound for a given K. The maximum of this bound calculation over all $0 \le K \le C/2$ is the lower bound L_2.

4.1 Adapting the Lower Bound to Partial Solutions

One simple way to apply the lower bound for a fixed capacity bin packing problem is to do so before search begins. If L_2 returns a value greater than the number of bins available, then we can declare the problem infeasible. However, we would like to benefit from the lower bound test when bins are of different sizes, meaning that it could be applied during search as well as treating problems that have bins of variable sizes from the outset.

The idea is that after each extension of the partial assignment of items to bins, we transform the current partial assignment into one which can be handled by the bounding function L_2. We then prune the search if the lower bound calculated exceeds the number of bins available.

The transformation carried out is relatively straightforward. First, we find the bin with the maximum potential load; this will then become the fixed capacity in the transformed problem: $C = \max_{j \in B} \overline{l_j}$. Then, in order to account for items already packed as well as bin capacities less than C, we create a new vector z of item sizes that contains the sizes of all item in U, the unpacked items, plus one additional item a_j for each bin j. We define:

$$a_j = p_j + C - \overline{l_j}$$

This requires some explanation as it involves two manipulations. First, we are introducing items into the packing problem representing the already packed items. We could just add the items themselves, but that would miss the opportunity to group items that have already been packed together. When we group these items, we have the potential for a better bound. Second, we are introducing items to 'top up' each bin with capacity less than C. That is, we have overestimated the capacity of bin j. So, to emulate the fact that a bin has a lesser capacity, we introduce an item into the transformed packing problem which says that an additional $C - \overline{l_j}$ must be packed. Finally, we take the further opportunity to fuse these two items into one as we know that all the packed items and the 'top up' item must be packed in the same bin.

We denote by a the vector $\langle a_1 \dots a_m \rangle$ and assume a function $\textsc{SortDec}(a)$ which sorts it non-increasing order. Likewise, we assume a function $\textsc{Merge}(x, y)$ which merges two such sorted vectors x and y, maintaining the ordering property. We now define the vector z of items sizes to be passed to L_2 as $z = \textsc{Merge}(\langle s_i \mid i \in U \rangle, \textsc{SortDec}(a))$. Search is pruned whenever $L_2(C, z) > m$. The time complexity of the bounding procedure is $O(n + m \log m)$.

5 Experiments

We conduct experiments to compare $\mathcal{P}$ to the typical model presented in section 2.4. Throughout these tests we use a machine with a Pentium IV processor running at 2GHz and 1GB of memory. Tests are performed with ILOG Solver.

We compare the typical model, which we call the *basic* level of propagation described in section 2.5, with the enhanced model incorporating the additional

rules described in sections 3 and 4. We decompose these extra rules into use of the lower bound described in section 4 (which we term "+LB"), and use of the pruning and propagation rules described in section 3 (which we term "+P"). Use of both of these sets of rules together is indicated by "+LB +P".

We use a standard search procedure, *complete decreasing best fit* (CDBF) [5], which packs items in order of non-increasing size, packing each item in the first bin with least free space that will accommodate it. On backtracking, the search states that the chosen item cannot be placed in the selected bin. A symmetry breaking rule is also used which states that on backtracking, all "equivalent" bins are also eliminated as candidates for the current item and indeed all other unpacked items of the same size. An equivalent bin is one which carries the same load as the one being eliminated from consideration. In addition, no choice point is created if all available bins for a particular item are equivalent: the item is packed into the first of them. Finally, we added the additional dominance rule which states that if an item can fill a partially filled bin to capacity, then it is immediately packed in this bin. This rule subsumes Gent's pair packing preprocessing rule [6], and is the simplest special case of a rule described in [11].

In CDBF, to find the minimal number of bins, we solve a succession of decision problems starting with the maximum number of bins set to the simple lower bound L_1. Each time the packing problem is proven to be insoluble, the number of bins is increased until a solution is found, which is necessarily optimal.

In the first instance, we chose a set of known benchmarks which could be comfortably solved by all models, from *basic* to +LB +P, so that comparisons could be made without extensive CPU times being expended. In this regard, we examined the smallest instances of series 1 of [15]. These problems have 50 items with sizes chosen from $\{[1, 100], [20, 100], [30, 100]\}$ and bin capacities chosen from $\{100, 120, 150\}$. The suite comprises 20 instances for each combination of capacity and size distribution, making 180 instances in total.

Figure 3 plots the number of choice points taken for the 180 benchmarks, where the benchmarks have been sorted according to the number of choice points needed to solve the problem using all additional propagations (+LB +P). In accordance with observations in [5, 10], there is great variation in the effort needed to solve the problems. For some problems, the additional propagation reduces the number of choice points by over three orders of magnitude. One problem took over three million choice points using the *basic* level, but was solved without branching using +LB +P.

Figure 4 plots two different views of the problems where additional propagations (+P) and lower bound pruning (+LB) are activated independently. We can see that the use of the additional propagations (+P) is more important than the use of the lower bound (+LB). In fact, the lower bound often performs no additional pruning over +P. However, on some problems, the search is significantly cut by the used of the lower bound. We argue that the small additional lower bound cost is paid back by increased robustness.

Table 1 shows all instances that were not solved in under 100 choice points. As mentioned, we can see that the lower bound calculation often performs no or

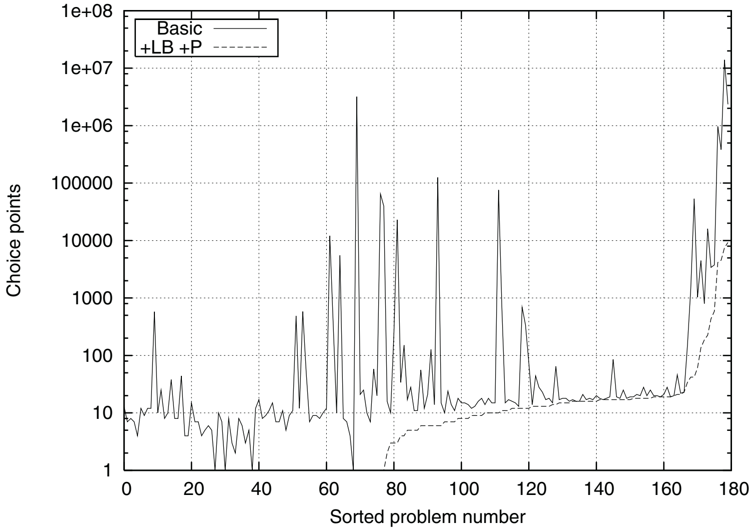

Fig. 3. Sorted 50 item instances

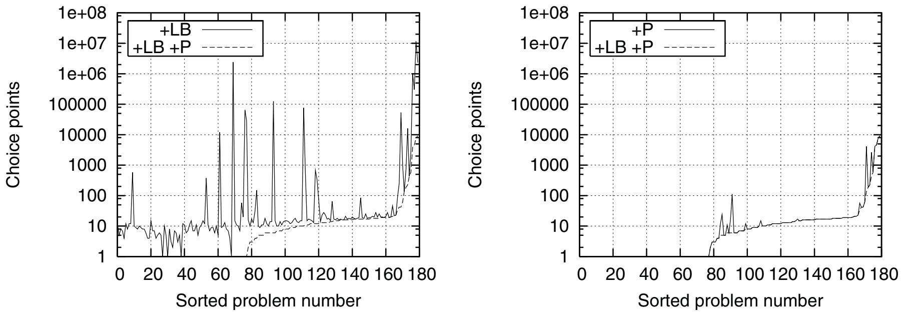

Fig. 4. Sorted 50 item instances, broken down into different propagation types

Table 1. Hardest instances, sorted by choice points needed by +LB +P

	Choice Points				Run Time (s)			
Name	Basic	+LB	+P	+LB +P	Basic	+LB	+P	+LB +P
N1C2W1_G	4485	138	4133	136	0.38	0.02	0.48	0.02
N1C3W2_G	802	742	187	187	0.05	0.05	0.03	0.03
N1C3W2_J	16101	16035	225	225	0.88	0.76	0.04	0.04
N1C2W1_C	3412	435	2639	435	0.22	0.04	0.24	0.05
N1C3W1_R	3756	2902	609	586	0.25	0.18	0.10	0.09
N1C1W2_A	966684	966684	4154	4154	42.29	38.00	0.40	0.44
N1C3W2_H	382812	302637	4562	4562	15.74	10.89	0.62	0.66
N1C3W2_F	13971619	11671895	7491	7491	651.95	483.19	1.45	1.53
N1C3W4_I	2354291	2342765	9281	9281	105.67	92.60	0.74	0.80

little pruning, but it did reduce the number of choice points by over a factor of thirty on one of these problems. The additional propagations (+P) can result in massive speed increases however, reducing run times by orders of magnitude.

We now look at another benchmark set in order to compare $\mathcal{P}$ with other other bin packing algorithms. We examine the benchmarks of Falkenauer [3]. These benchmarks are well-known, but have been criticized by Gent [6] as being too easy. The problems can certainly be solved by simple methods. However, Gent's methods are geared towards consideration of *bins*, and finding items which fill them to completely to capacity; they do not resemble CDBF, which might be applied as a first try at solving a problem with a bin packing component, before exploring less well-known methods. Falkenauer gives the performance of Martello and Toth's branch and bound algorithm [12] on these problems which was, until more recently, the best exact algorithm for bin packing. This method is also based on the decreasing best fit strategy, but makes heavy use of quite complex reduction procedures, lower bounds and dominance criteria.

Table 2 shows the results of runs on the smallest sizes of the "uniform" and "triplets" benchmarks using all propagations (+LB +P). Comparison of run times should be done with care, as the results of Martello and Toth's algorithm (MTP) are reproduced from [3] and are likely to be around one order of magnitude greater than those of $\mathcal{P}$. However, the number of choice points is a fairly reliable measure. Where MTP is marked with a > sign, it means that the optimum solution was not found. All problems are solved fairly easily by the combination of $\mathcal{P}$ and CDBF except for `u120_08` and `u120_19`. The former is solved in under 7 minutes, whereas the latter takes 15 hours. Martello and Toth's procedure falls foul of problems `u120_08` and `u120_19` as does ours, but performs much worse on the triplets set, finding optima for only 6 of the problems. By contrast, $\mathcal{P}$ plus CDBF solves all of these problems quickly.

Korf [11] also tested his algorithm on these problem sets, finding that his algorithm could solve all problems quickly. However, one interesting phenomenon is that for the two problems that his algorithm found the most difficult, our method found solutions instantly. Conversely, for the two most difficult problems we report, his algorithm found solutions instantly. The fact that MTP and CDBF find uniform problems 08 and 19 difficult, while Korf's algorithm does not, seems to indicate that a change of search strategy could be useful. This is supported by the fact that Gent's methods [6] were quickly successful on these problems. Korf also tested on the triplets set with 120 items and reported finding solutions instantly to all but four of the twenty problems. We do not report our results on these problems for reasons of space, but interestingly, our algorithm found solutions to all of these four troublesome problems instantly.

What is perhaps surprising about these experiments is that although constraint programming is a general technology which is rarely the best method to apply to a pure problem, by a combination of a dedicated constraint and a standard search procedure, we manage to produce a competitive algorithm.

Table 2. Falkenauer's problems solved with +LB +P, comparison with MTP

	Choice Points		Run Time (s)	
Name	MTP	$\mathcal{P}$	MTP	$\mathcal{P}$
u120_00	56	39	0.1	0.02
u120_01	0	36	0.1	0.01
u120_02	124935	38	29.0	0.02
u120_03	74	31	0.0	0.02
u120_04	0	38	0.0	0.02
u120_05	43	32	0.1	0.02
u120_06	69	32	0.0	0.04
u120_07	54	38	0.0	0.03
u120_08	> 10M	2.63M	> 3681.4	398.34
u120_09	103	35	0.1	0.03
u120_10	0	34	0.1	0.01
u120_11	64	32	0.1	0.02
u120_12	88	25	0.0	0.03
u120_13	0	34	0.0	0.02
u120_14	0	33	0.0	0.02
u120_15	36	36	0.1	0.02
u120_16	0	33	0.0	0.02
u120_17	48	30	0.0	0.02
u120_18	24	35	0.0	0.01
u120_19	> 7.5M	321.89M	> 3679.4	15H08

	Choice Points		Run Time (s)	
Name	MTP	$\mathcal{P}$	MTP	$\mathcal{P}$
t60_00	36254	62	9.5	0.03
t60_01	28451	173	12.6	0.05
t60_02	> 1.5M	116	> 564.2	0.02
t60_03	> 1.5M	195	> 444.7	0.04
t60_04	> 1.5M	12	> 404.6	0.01
t60_05	> 1.5M	176	> 415.2	0.04
t60_06	> 1.5M	77	> 485.7	0.02
t60_07	> 1.5M	193	> 395.9	0.04
t60_08	> 1.5M	359	> 451.6	0.06
t60_09	26983	201	9.6	0.06
t60_10	1783	16	0.9	0.01
t60_11	13325	36	6.3	0.01
t60_12	6450	24	1.5	0.01
t60_13	> 1.5M	30	> 385.0	0.01
t60_14	> 1.5M	14	> 400.8	0.01
t60_15	> 1.5M	90	> 537.4	0.02
t60_16	> 1.5M	30	> 528.3	0.01
t60_17	> 1.5M	50	> 429.9	0.01
t60_18	> 1.5M	146	> 385.6	0.03
t60_19	> 1.5M	140	> 399.5	0.03

6 Related Work

Johnson's initial work [8] brought bin packing algorithms to the fore, but exact algorithms were lacking until the Martello and Toth's reduction and branch and bound procedures [12, 13]. However, in the last five years there has been significant interest and progress from the OR community (for example [1, 15]) using branch and bound and mathematical programming approaches. Korf's bin completion algorithm [10, 11], which makes heavy use of dominance properties, and Gent's work [6] have shown that the AI field has much to offer the domain. Trick [17] has shown that GAC can be achieved for knapsack constraints – a key subproblem of bin packing – using dynamic programming. Sellmann [16] has further proposed approximating this consistency to accelerate solving procedures.

7 Conclusion

This paper has introduced a constraint for bin packing. The constraint uses pruning and propagation rules based on a notion of *neighboring subsets* in subset sum problems as well as a lower bound on the number of bins used. We have demonstrated that this new constraint can cut search by orders of magnitude. Additional comparisons on established benchmarks showed that the new constraint coupled with a simple standard packing algorithm can significantly outperform Martello and Toth's procedure.

The packing constraint $\mathcal{P}$ has been available in ILOG Solver since version 6.0. All propagations described here will be available in Solver 6.1 to be released in autumn 2004.

References

1. J. Valerio de Carvalho. Exact solution of bin-packing problems using column generation and branch-and-bound. *Annals of Operations Research*, 86:629–659, 1999.
2. E. G. Coffman, Jr, M. R. Garey, and D. S. Johnson. Approximation algorithms for bin packing: A survery. In D. Hochbaum, editor, *Appoximation algorithms for NP-Hard Problems*, pages 46–93. PWS Publishing, Boston, 1996.
3. E. Falkenauer. A hybrid grouping genetic algorithm for bin packing. *Journal of Heuristics*, 2:5–30, 1996.
4. D. K. Friesen and M. A. Langston. Variable sized bin packing. *SIAM Journal on Computing*, 15:222–230, 1986.
5. I. Gent and T. Walsh. From approximate to optimal solutions: Constructing pruning and propagation rules. In *Proceedings of the 15th IJCAI*, 1997.
6. I.P. Gent. Heuristic solution of open bin packing problems. *Journal of Heuristics*, 3:299–304, 1998.
7. ILOG S.A., Gentilly, France. *ILOG Solver 6.0. User Manual*, September 2003.
8. D. S. Johnson. Fast algorithms for bin packing. *Journal of Computer and System Sciences*, 8:272–314, 1974.
9. H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, 2004.
10. R. Korf. A new algorithm for optimal bin packing. In *Proceedings of 18th AAAI*, pages 731–736, 2002.
11. R. Korf. An improved algorithm for optimal bin packing. In *Proceedings of the 18th IJCAI*, pages 1252–1258, 2003.
12. S. Martello and P. Toth. *Knapsack problems*. Wiley, Chichester, 1990.
13. S. Martello and P. Toth. Lower bounds and reduction procedures for the bin packing problem. *Discrete and Applied Mathematics*, 28(1):59–70, 1990.
14. J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the 12th AAAI*, pages 362–367. American Association for Artificial Intelligence, AAAI Press / The MIT Press, 1994.
15. A. Scholl, R. Klein, and C. Jürgens. BISON: A fast hybrid procedure for exactly solving the one-dimensional bin packing problem. *Computers & Operations Research*, 24:627–645, 1997.
16. M. Sellmann. Approximated consistency for knapsack constraints. In *Proceedings of the CP' 03*, pages 679–693. Springer, 2003.
17. M. Trick. A dynamic programming approach for consistency and propagation for knapsack constraints. In *Proceedings of CP-AI-OR '01*, 2001.